Practical Work Part 2: WIMP application

This section is dedicated to the development of a *WIMP* application (Window, Icons, Menus, Pointing device), i.e. an application with a *GUI* (Graphical User Interface) based on basic clickable interactive objects (icons, menus, buttons…). **duration: 4h**

Objectives:

- Assemble nodes in a rich layout,
- Subscribe and react to a rich set of events,
- Use a few design patterns,
- Architect code to separate functional core from visualization/interaction.

Documentation:

JavaFX User Interface components
http://docs.oracle.com/javase/8/javafx/user-interface-tutorial/index.html

JavaFX Layouts
http://docs.oracle.com/javase/8/javafx/layout-tutorial/index.html

JavaFX Events processing
http://docs.oracle.com/javase/8/javafx/events-tutorial/processing.htm

JavaFX Javadoc
https://docs.oracle.com/javase/8/javafx/api/toc.htm

Exercise 1: Tic-tac-toe game

1.1 Preparation

Launch *NetBeans* and create a new project named "PracticalWork2" without main class (uncheck the box "Create Application Class"). Copy then Paste (Paste > Copy contextual menu) the code and resources provided with this subject on e-campus, according to the following indications:

- Paste the content of the folder "src" in your project "Source Packages" folder,
- paste the folder "resources" at the root of your project folder (it contains images to be displayed).

The provided code is a tic-tac-toe game with two fully functional implementations, in a textual mode and with a *GUI* developed in *Swing*, one of the historical graphic libraries in *Java*. In this exercise you will make a third version in *JavaFX*. And you will do it easily because the code is architected following a certain number of ***design patterns***, i.e. methods/good practices that solve general software problems and thus facilitate maintainability and scalability.

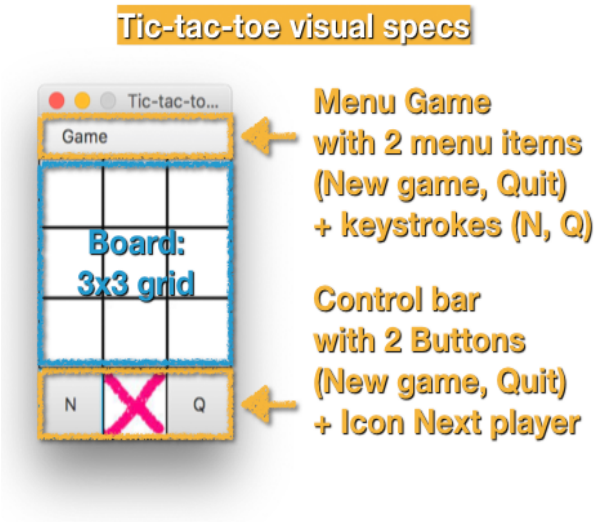
After testing that the application works well on your computer read through the code and it's documentation and examine the class and sequence diagrams provided in the appendices, then answer the following questions:

- What are the names of the design patterns used in this project?
- What are they used for?
- Which one allows to easily switch between the different versions of the *GUI*?
- Which one will most facilitate a change in technology (the development of our *JavaFX* version)?

We are now going to develop the *JavaFX* version in several steps: reproduce the game layout of the *Swing* version, add basic interactivity, link it to the functional core, enhance usability by adding menus, keystrokes…

To help you a class logically named `TicTacToeJavaFXView` is provided, with a lot of things already set up. Take the time to read it to revise the structure of a *JavaFX* application and to understand the elements specific to our game.

1.2 Layout



The root container of the game is a `VBox`, which means it will align its children vertically according to the visual specifications above. Identify in the *JavaFX* Layouts documentation the best candidates to compose the board and the control bar.

Because it is simpler you can begin by creating the control bar:

- create a suitable container to hold the control bar,
- add it in the root container,
- create the buttons (use `Button` instances),
- create the next player icon (use an `ImageView` instance displaying the `Image` named `VOID`),
- don't forget to add the 3 widgets to their common parent.

Test your app (don't forget to switch the view in the main class factory) and fix the alignment problems. To do so you'll have to set the buttons the same size as the icon, look for the appropriate method in the documentation.

Now let's move on to the game board:

- create the container to hold the squares and add it to the root container,
- use `ImageView` instances for the squares,
- create them intelligently (the board size could be different from one game to another),
- add the squares to the array named `squareViews` (we will need to keep a reference to them),
- add the squares to their parent (there is another specific way to do it, search the documentation).

We now have the same general appearance as in the *Swing* view, except for the menu bar, which we will add later. We can consider we've finished to code the *view* aspect of our *JavaFX* version. Let's add a bit of interactivity!

1.3 Interactivity

Begin with the buttons, you already know how to do it. The reaction to actions on a button will be in a first time to write an intelligible message in the console.

For the squares we will adopt another strategy: factoring the `EventHandler` rather than creating one for each square. In this unique `EventHandler` you will retrieve the source of the event to identify the square and look for its indices in the `squareViews` table we now need. Then you will print them in the console.

We have almost completed the development of the *controller* part. The only thing left to do to make the game functional is to link it to the *functional core*.

1.4 Link to the *functional core*

Replace the prints in the console with calls to the right methods in the model. The *model* will then call the right methods of the *view* to update it according to the results of its calculations (see sequence diagram).

To be able to do so the very last step is to implement the remaining empty methods in the *view*. You can take inspiration from the *Swing* version and use instances of `Alert` instead of calls to the *Swing* specific `JOptionPane.showMessageDialog()` class method.

Your *JavaFX* version is now functional. The last question is dedicated to the implementation of a few standard services that improve the usability of an interactive application and are considered mandatory.

1.5 Usability enhancements

Even though in our game all actions are directly accessible (squares and buttons to click) we are going to learn how to set up **menus**. Implement your menu by following these steps:

- create a `MenuBar` instance,
- add it as the first child of the root container (search the `java.util.List` documentation to insert an element at a specified position),
- create a `Menu` instance and add it as a child of the `MenuBar`,
- create a `MenuItem` instance and add it as a child of the `Menu`,
- subscribe to `ActionEvent` on the `MenuItem`.

As we want the same reaction to the same event on a `Button` and a `MenuItem`, factor it into a single `EventHandler` per function or even better, a single one for all.

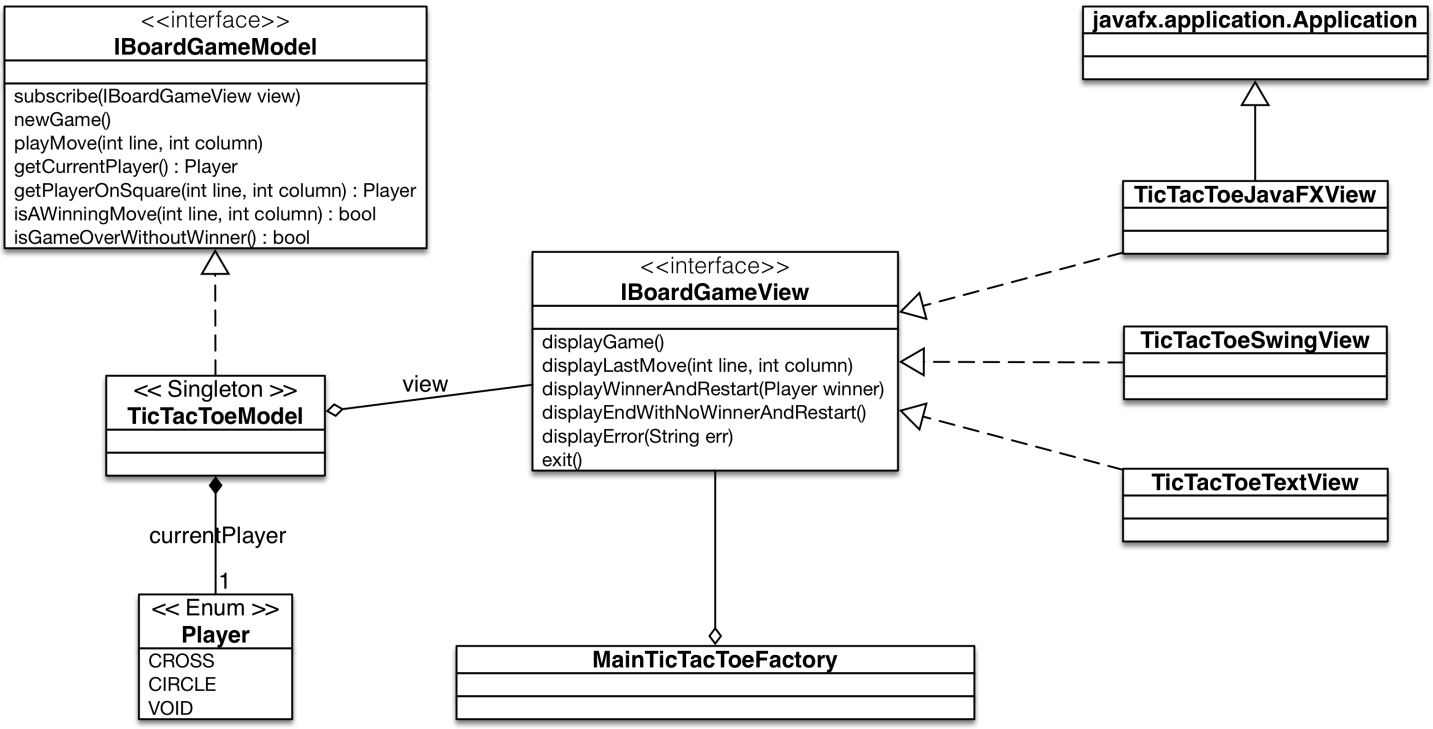
In addition to the menus, at least the most important functions should also be accessible via keyboard **shortcuts** (or accelerators):

- create a `KeyCombination` instance,
- associate it to the chosen `MenuItem` thanks to its `setAccelerator()` method.

There are still a lot of improvements to be made to our game. We don't have the time to realize them, but let's take a moment to discuss them before we leave. What could be these improvements?

Congratulations! You now know how to design and develop a standard *WIMP* application. In Practical Work Part 3 we will learn to design and develop *direct manipulation* interactions such as *pan, drag, mouse centered zoom*… See you soon!

Appendix 1: Class Diagram



Appendix 2: Sequence Diagram